Retrograde Analysis of some Chinese Chess Endgames

Wu, Ren; Beal, Don

ISSN 1369-1961

Department of
Computer Science

Technical Report No. 731

Retrograde Analysis of some Chinese Chess Endgames

Ren Wu &
Don Beal

QUEEN MARY
AND WESTFIELD COLLEGE
UNIVERSITY OF LONDON

December 1996

Retrograde Analysis of some Chinese Chess Endgames

Department of Computer Science, Queen Mary and Westfield College
Technical report 731, December 1996

Ren Wu (ren@dcs.qmw.ac.uk)
Don Beal (don@dcs.qmw.ac.uk)
Department of Computer Science
Queen Mary and Westfield College
Mile End Road
London E1 4NS

Abstract

In this technical report, we present a fast, memory-efficient retrograde algorithm which can be used to generate chess/Chinese chess endgame databases. It only requires 1 bit per position of RAM, while still producing the full depth information in the final file on disc. In comparison, simpler algorithms need either a byte per position in RAM, or produce only win/draw information, or slow down tremendously swapping parts of the data to and from disc. A nine-pieces Chinese chess endgame has been constructed by this algorithm in less than 20 hours on a Pentium PC. The authors estimate that the algorithm would produce a five men pawn-less (western) chess endgame database in less than a day on a 100Mhz Pentium PC. The algorithm also makes it feasible to produce 6-men chess endgame databases on a modern workstation.

1. Introduction

The construction of endgame databases has several benefits. First, it gives us a piece of truth about the game. Second, the databases, because of their perfect knowledge about certain domains, are a useful background for Artificial Intelligence research, especially in machine learning. Third, in some games. the databases will significantly increase the strength of game-playing programs. Many endgame databases in many games have been constructed since Ströhlein's (Ströhlein, 1970) pioneer work. Thompson (Thompson, 1986) generated almost all 5-men chess endgame databases and made them widely available in CD format. Stiller (Stiller, 1995) generated all 5-men pawn-less chess endgame databases and a few 6-men chess endgame databases on a supercomputer. Edwards has constructed many 5-men chess endgame databases (based on distance-to-mate instead of distance-to-conversion) and made them publicly available via Internet. Perhaps the most impressive endgame database construction so far is Schaeffer's work (Schaeffer et al, 1994) on Checkers. His program created all 8 men endgame databases and some of the 9 men databases. The databases contained more than 440 billion(4.4×10^{11}) positions. The databases played a very important role in Chinook's success.

However, all the algorithms demand large computing resources. Thompson's (1986) algorithm needs two bits of RAM for every board configuration (32MB for 5-men endgames), or random access to disc files. Stiller's algorithms need a highly-parallel supercomputer. Others also need similarly large amounts of memory or need a very long time to build a moderate database. In chess for example, it has been reported that a 5-man endgame database can take 89 days on a 486 PC. (S. J. Edwards, 1996).

In this report, we describe a significantly more memory-efficient retrograde algorithm that we developed during our research on constructing Chinese chess endgame databases. The new algorithm is better than our previous one in three important ways. First of all, it requires only half the random access memory and is faster. Secondly, it keeps full information about the endgame. Thirdly, it can calculate either distance-to-conversion or distance-to-win/loss.

2. Previous Work

The basic retrograde algorithm was first described by Ströhlein (Ströhlein, 1970). It was independently re-invented by several other people, such as Clark (Clark 1977) and Thompson (Thompson, 1986). Herik and Herschberg(1985) give a tutorial introduction to the retrograde concept. Thompson' s paper also gives a brief but clear outline of his algorithm.

The algorithms all assume one side is stronger than another and only calculate the distance to win, which is the number of moves the stronger side needs to either to mate the weaker side or transfer into a known win position in a subgame. There are two particular disadvantages:

- the algorithm assumes that if a position the stronger side is not win, it must be a draw. This is not always true, and so a separate database needs to be constructed for the other side.
- Sometimes the distance to mate may be more appropriate than the distance to conversion.

To address these problems, we designed an algorithm which keeps both side's full information when generating Chinese chess endgame databases. Edwards (Edwards, 1996) also designed a algorithm which can generate chess endgame database containing both side's distance to mate/loss. However, these algorithms are slower than Thompson's because of the even larger memory requirement.

In the following sections, we will take a close look to these algorithms.

2.1 Ken Thompson

Thompson's algorithm can be formulated as follows:

```
BITS    W, B, b, w, j;
DATABASE db;

void    TopLevel(void)
{
        int     n = 0;

        P1(b);
        while (!done)
        {
                P2(b, w, n);
                n++;
                P3(w, j, n);
                P4(j, b, n);
        }
}

// Random:          b0
// Sequential:

/* P1() do
** for every godel index
**      1. convert it into position, assume it is BTM
**      2. if black has been mated, set bit in b
** b then is a bitmap of all the BTM and mated postions.
*/
void    P1(BITS b)
{
        /* scan all positions, assum BTM */
        for (b = null, g = 0; g < maxGodel; g++)
        {
                if (Mated(GodelToPos(BTM, g)))
                        b[g] = 1;
        }

        B = b;
        W = null;
}

// Random:          W, wi+1
// Sequential:      bi
/* P2()
```

```
**      given a bitmap of BTM and lose in n positions, P2 set the bitmap of
**      all WTM and win_in_n+1 positions.
**      b - bitmap of BTM and lose_in_n.
**      w - bitmap of WTM and win_in_n+1.
**      n - distance.
*/
void    P2(BITS b, BITS w, int n)
{
        w = null;

        for (g = 0; g < maxGodel; g++)
        {
                if (b[g])
                {
                        nPred = GenPredPos(GodelToPos(BTM, g), posPred[]);
                        for (i = 0; i < nPred; i++)
                        {
                                gPred = PosToGodel(posPred[i]);
                                if (!W[gPred])
                                        w[gPred] = 1;
                        }
                }
        }

        W |= w;

        for (g = 0; g < maxGodel; g++)
                if (w[g])
                        db[g] = n+1;
}

// Random:      B, j
// Sequential:  w
/* P3()
**      given the bitmap of WTM and win_in_n positions, P3() generate the
**      bitmap of BTM and black may lose_in_n positions.
**      w - WTM win_in_n;
**      j - BTM maylose_in_n;
**      n - distance;
*/
void    P3(BITS w, BITS j, int n)
{
        j = null;

        for (g = 0; g < maxGodel; g++)
        {
                if (w[g])
                {
                        nPred = GenPredPos(GodelToPos(WTM, g), posPred[]);
                        for (i = 0; i < nPred; i++)
                        {
                                gPred = PosToGodel(posPred[i]);
                                if (!B[gPred])
                                        j[gPred] = 1;
                        }
                }
        }
}

// Random:      W, b
// Sequential:  j
/* P4()
**      given the bitmap of BTM and maylose_in_n positions, P4() calulate
**      the bitmap of BTM and lose_in_n positions.
**      j - BTM and maylose_in_n;
**      b - BTM lose_in_n;
**      n - distance;
*/
void    P4(BITS j, BITS b, int n)
{
        b = null;
        for (g = 0; g < maxGodel; g++)
        {
                if (j[g])
                {
                        nSucc = GenSuccPos(GodelToPos(BTM, g), posSucc[]);
                        for (i = 0; i < nSucc; i++)
                        {
                                gSucc = PosToGodel(posSucc[i]);
                                if (!W[gSucc])
                                        break;
                        }
                        if (i == nSucc)             // all succ pos are in W
                                b[g] = 1;
                }
        }
```

```
        B += b+1;
}
```

In his original implementation , he used files to represent the database and the bitmaps. On a Sequent Balance 8000 computer with 12 National 32032 cpus and 16 MB shared memory, a typical 5-men pawn-less chess endgame would be solved in about three weeks real time.

Of course, using files as the prime data structure is not optimal and the algorithm will benefit from RAM if available. Let us examine what is the memory requirement for this algorithm.

P1 only accesses one bitmap and the access is sequential.
P2 accesses three bitmaps and the database itself. P2 can be divided into two parts. In the first part, it needs sequential access to b and random access to W and w. For the second part, it needs sequential access both w and W.
P3 also accesses three bitmaps, namely w, j, B. It uses sequential access for w and random access for j and B.
P4 once again access three bitmaps, j, b and W. It uses sequential access for j, b and random access for W.

Bitmaps that are accessed sequentially can be read from disc efficiently. They can be processed as disc files without reducing runtime efficiency very much. However, if random access to bitmaps on disc is attempted, there will be a severe loss of speed. The peak requirement here is memory for the two randomly-accessed bitmaps during P3 and P4. These dominate the memory required for efficient operation, and total about ¼ of the size of the database. For example, full 5-men pawn-less endgames are about 120MB, so we need about 30 MB memory.

2.2 Wu and Beal

Back in 1992, we started work on construction of Chinese chess endgame databases. The main goal of this work was to set up a "production line" which could generate any Chinese chess endgame database without rewriting or recompiling the program, and which had sufficient consistency checks and tests to give confidence that the results were error-free. We also chose to produce both side's win/draw/lose result in integrated fashion instead just one side's win/draw for any one database. The algorithm, and the results will be presented in the first author's forthcoming Ph.D. thesis.(Wu), but here is a simplified description:

```
DATABASE wdb, bdb;
BITS     maylose;

int      w_win[], b_win[];

TopLevel()
{
        DoInitialize();

        n = 1;
        while (DoWinInN(n))
                n++;
}

int      DoWinInN(int n)
{
        if (w_win[n] > 0)
        {
                DoMaylose(BLACK, wdb, maylose, n);
                DoLose(BLACK, maylose, bdb, wdb, n);
                w_win[n+1] = DoWin(WHITE, bdb, wdb, n+1);
        }

        if (b_win[n] > 0)
        {
                DoMaylose(WHITE, bdb, maylose, n);
                DoLose(WHITE, maylose, wdb, bdb, n);
                b_win[n+1] = DoWin(BLACK, wdb, bdb, n+1);
        }

        return(w_win[n+1] + b_win[n+1]);
}
```

```
DoMaylose(int color, DATABASE db1, DATABASE db2, BITS maylose, int n)
{
        /*      1. for all position p, db1[p] == n
        **      2. generate its predecessor positions pp
        **      3. set bit in maylose for every pp if db2[pp] == unknown
        */
}

DoLose(int color, BITS maylose, DATABASE db1, DATABASE db2, int n)
{
        /*      1. for all position p in maylose
        **      2. generate its successor positions sp
        **      3. db1[p] == lose_in_n if for every sp, db2[sp] == WIN
        */
}

DoWin(int color, DATABASE db1, DATABASE db2, int n)
{
        /*      1. for all position p in db1, if db1[p] == lose_in_n
        **      2. generate its predecessor positions pp
        **      3. for every pp, db2[pp] == win_in_n+1 if db2[pp] == unknown
        */
}
```

Let us take a look what is the memory requirement for this algorithm.

Both TopLevel() and DoWinInN() are high level function and just make calls to low level functions. They are not using extra memory.
DoMaylose() uses random access for bitmap *maylose* and database *db2*. It also uses sequential access for database *db1*.
DoLose() uses random access for database *db2* and sequential access for bitmap *maylose* and database *db1*.
DoWin() uses random access for database *db2* and sequential access for database *db1*.

The peak requirement here is function DoMaylose(), which require one bitmap and one database been randomly accessed, or 9 bits per position. For example, for full 5-men pawn-less endgame, we need 120+120/8MB, or 135MB RAM!

Because the size of the databases, we were unable to put them all in memory on the workstations we were using. In our first implementation, we used a similar technique to that used by Schaeffer etc. (Schaeffer etc. 1994) to cache the database in memory. But even with a good disk cache technique, the program still spends a lot of time on disk access and is usually I/O bounded.

2.4 Steven J. Edwards

Although we don't have the details of Steven's algorithm, we can guess that he uses similar techniques to ours to cache the databases. The algorithm is very slow: a five men chess endgame database took more than 89 days to build on a 486 PC. He is facing the same difficulty we had. Here is a paragraph from his recent article (Edwards, 1996),

> "Why, a reader might ask, did the run take so much time? Given that KBBKN WTM had been generated on a micro in about 200 hours (about 11 times faster), this is a reasonable question. The answer is that the algorithm used and the results obtained are different. Because SPECTOR calculates exact distance-to-mate/loss numbers instead of distance-to-conversion, it is necessary for it to manipulate long vectors of integers (bytes) instead of the eight times more compact vectors of bits. This requirement is caused by the need to "fold in" earlier distance-to-mate/loss numbers from successor databases. The bit-vector approach used for generating distance-to-conversion databases needs only to know if a successor databases position is won or not-won, while SPECTOR must have the exact number of moves. Additionally, SPECTOR always generates both WTM and BTM sides of an endgame instead of only a single side. Finally, SPECTOR's results distinguish

among mate/loss/draw verdicts while most other databases combine the drawn and lost positions into a single category."

As you can see from the next section, our newer algorithm produces the same data as SPECTOR and, at the same time, improves on memory requirements compared to Thompson's. We also estimate it to be faster than both , but there is no directly comparable data as they haven't been run on the same tasks.

3. The one-bitmap algorithm

We present our one-bitmap algorithm in a pseudo C format:

```
DATABASE wdb, bdb;          // byte streams, squential access only
SBITS    s_bits;            // squential access bit stream
RBITS    r_bits;            // random access bit stream

int      w_win[], w_lose[], b_win[], b_lose[];

int      distance_to_mate;

void     TopLevel(void)
{
         int      n;

         DoInitialize();

         DoWinInOne();

         n = 1;
         while (DoWinInN(n))
                  n++;
}

void     DoInitialize(void)
{
         /*       DoInitialize() scan all positions and do three things
         **                1.       if a position is illegal, mark it as illegal.
         **                2.       if a position is acturaly in a subgame, load the subgame
         **                         database result if we use distance_to_mate, or
         **                         mark it as win_in_0, or lose_in_0, depend on the result
         **                         if we use distance_to_conversion.
         **                3.       if it is WTM and white can capture black's king, mark it
         **                         as win_in_0 in wdb. Same for the BTM.
         */
}


void     DoWinInOne(void)
{
         /*       WTM, Lose_In_0 */
         GetBits(s_bits, wdb, DB_UNKNOWN);                  // s_bits = WTM unknown
         if (distance_to_mate)                              //
                  GetBits(r_bits, bdb, DB_WIN_IN_N(0));     // r_bits = BTM win_in_0
         else                                               //
                  GetBits(r_bits, bdb, DB_WIN);             // r_bits = BTM wins
         DoLose(WHITE, s_bits, r_bits);                     // s_bits = WTM lose_in_0
         w_lose[0] += AddBits(wdb, s_bits, DB_LOSE_IN_(0));

         /*       BTM Lose_in_0 */
         GetBits(s_bits, bdb, DB_UNKNOWN);                  // s_bits = BTM unknown
         if (distance_to_mate)                              //
                  GetBits(r_bits, wdb, DB_WIN_IN_N(0));     // r_bits = WTM win_in_0
         else                                               //
                  GetBits(r_bits, wdb, DB_WIN);             // r_bits = WTM wins
         DoLose(BLACK, s_bits, r_bits);                     // s_bits = BTM lose_in_0
         b_lose[0] += AddBits(bdb, s_bits, DB_LOSE_IN_(0));

         /* WTM, Win_In_1 */
         GetBits(s_bits, bdb, DB_LOSE_IN_(0));              // s_bits = BTM lose_in_0
         ClearBits(r_bits);                                 // r_bits = empty
         DoMayWin(WHITE, s_bits, r_bits);                   // r_bits = WTM maywin
         GetBits(s_bits, wdb, DB_UNKNOWN);                  // s_bits = WTM unknown
         AndBits(s_bits, r_bits);                           // s_bits = WTM win_in_1
         w_win[1] = AddBits(wdb, s_bits, DB_WIN_IN_(1);

         /* BTM, Win_In_1 */
         GetBits(s_bits, wdb, DB_LOSE_IN_(0));              // s_bits = WTM lose_in_0
         ClearBits(r_bits);                                 // r_bits = empty
         DoMayWin(BLACK, s_bits, r_bits);                   // r_bits = BTM maywin
         GetBits(s_bits, bdb, DB_UNKNOWN);                  // s_bits = BTM unknown
```

```
        AndBits(s_bits, r_bits);                        // s_bits = BTM win_in_1
        b_win[1] = AddBits(bdb, s_bits, DB_WIN_IN_(1));
}

int     DoWinInN(int n)
{
        if (w_win[n] > 0)
        {
                GetBits(s_bits, wdb, DB_WIN_IN_(n));        // s_bits = WTM win_in_n
                ClearBits(r_bits);                          // r_bits = empty
                DoMayLose(BLACK, s_bits, r_bits);           // r_bits = BTM maylose or unknown
                GetBits(s_bits, bdb, DB_UNKNOWN);           // s_bits = BTM unknown
                AndBits(s_bits, r_bits);                    // s_bits = BTM maylose

                if (distance_to_mate)
                        GetBits(r_bits, wdb, DB_WIN_IN_(n));        // r_bits = WTM win_in_n
                else
                        GetBits(r_bits, wdb, DB_WIN);       // r_bits = WTM wins
                DoLose(BLACK, s_bits, r_bits);              // s_bits = BTM lose_in_n
                b_lose[n] = AddBits(bdb, s_bits, DB_LOSE_IN_(n));

                if (distance_to_mate)
                        GetBits(s_bits, bdb, DB_LOSE_IN_(n));       // s_bits = lose_in_n
                ClearBits(r_bits);                          // r_bits = empty
                DoMayWin(WHITE, s_bits, r_bits);            // r_bits = WTM maywin
                GetBits(s_bits, wdb, DB_UNKNOWN);           // s_bits = WTM unknown
                AddBits(s_bits, r_bits);                    // s_bits = WTM win_in_n+1
                w_win[n+1] = AddBits(wdb, s_bits, DB_WIN_IN_(n+1));
        }

        if (b_win[n] > 0)
        {
                if (r_win[n] > 0 || n == 1)
                        GetBits(s_bits, bdb, DB_WIN_IN_(n));        // s_bits = BTM win_in_n
                ClearBits(r_bits);                          // r_bits = empty
                DoMayLose(WHITE, s_bits, r_bits);           // r_bits = WTM maylose or unknown
                GetBits(s_bits, wdb, DB_UNKNOWN);           // s_bits = unknown
                AndBits(s_bits, r_bits);                    // s_bits = WTM maylose

                if (distance_to_mate)
                        GetBits(r_bits, bdb, DB_WIN_IN_(n));        // r_bits = BTM win_in_n
                else
                        GetBits(r_bits, bdb, DB_WIN);       // r_bits = BTM wins
                DoLose(WHITE, s_bits, r_bits);              // s_bits = WTM lose_in_n
                w_lose[n] = AddBits(wdb, s_bits, DB_LOSE_IN_(n));

                if (distance_to_mate)
                        GetBits(s_bits, wdb, DB_LOSE_IN_(n));       // s_bits = WTM lose_in_n
                ClearBits(r_bits);                          // r_bits = empty
                DoMayWin(BLACK, s_bits, r_bits);            // s_bits = BTM maywin
                GetBits(s_bits, bdb, DB_UNKNOWN);           // r_bits = BTM unknown
                AddBits(s_bits, r_bits);                    // s_bits = BTM win_in_n+1
                b_win[n+1] = AddBits(bdb, s_bits, DB_WIN_WIN_(n+1));
        }
}


void    DoMayLose(int color, SBITS wins, RBITS maylose)
{
        /*      DoMayLose() do
        **      for every bit set in wins,
        **              1.      get the index and  convert it to position
        **              2.      generate all predecessor positions
        **              3.      convert each predecessor back to index and set bit in maylose.
        */
}

void    DoLose(int color, SBITS maylose, RBITS wins)
{
        /*      DoLose() do
        **      for every bit set in maylose,
        **              1.      get the index and  convert it to position
        **              2.      generate all successor positions
        **              3.      if any successor in not in wins, clear this bit in maylose.
        */
}

void    DoMayWin(int color, SBITS loses, RBITS maywin)
{
        /*      DoMayWin() do
        **      for every bit set in loses,
        **              1.      get the index and  convert it to position
        **              2.      generate all predecessor positions
        **              3.      convert each predecessors back to index and set bit in maywin.
        */
}
```

```
void    GetBits(SBITS sbits, DATABASE db, int db_value)
{
        /*      GetBits() scan db and set bit in sbits if db's value meets db_value */
}

void    AddBits(DATABASE db, SBITS sbits, int db_value)
{
        /*      AddBits() set, for all positions in sbits, db's value as db_value */
}

void    ClearBits(SBITS sbits)
{
        /* Clear all bits */
}

void    AndBits(SBITS sbits, RBITS rbits)
{
        /* bitwise ADD */
}
```

The algorithm itself is quite straight forward and easy to understand. Let us analysis the memory requirment for this algorithm.

TopLevel() is a high level function and only call other low level functions.
DoIntialize() uses sequential access to database *wdb* and *bdb*.
Both DoWinInOne() and DoWinInN() use sequential access for bitmap *s_bits*, database *wdb* and *bdb*. It also uses random access for bitmap *r_bits*.
DoMaylose() uses sequential access for bitmap *wins* and random access for bitmap *maylose*.
DoLose() uses sequential access for bitmap *maylose* and random access for bitmap *wins*.
DoMayWin() uses sequential access for bitmap *loses* and random access for bitmap *maywin*.
GetBits() uses sequential access for bitmap *sbits* and random access for database *db*.
AddBits() uses sequential access for bitmap *sbits* and database *db*.
ClearBits() uses sequential access for bitmap *sbits*.
AndBits() uses sequential access for bitmap *sbits* and bitmap *rbits*.

Then the peak memory requirement is in DoWinInOne() and DoWinInN(), which use random access for one bitmap and sequential access for one bitmap and two databases. Because the sequential file access is many times faster than random access, we keep the random access bitmaps to a minimum, and require just one bitmap in memory. Apart from the chess/Chinese chess specific routines, the rest is just simple load and Boolean operations over the files.

Again as a example, for full 5-men pawn-less endgame, we only need 120/8, or 15MB RAM.

4. Results from Chinese chess

We give two examples here, more can be found from the first author's thesis. Both are 9-men Chinese chess endgame databases.

Examples 1 - KAGP:KAAEE

	Total positions	Database size	Maximum distance	Build time
Distance-to-conversion	2,455,818,750	152,806,500	32	19:30:42
Distance-to-mate	2,455,818,750	152,806,500	47	28:13:30

Examples 2 - KEGP:KAAEE

	Total positions	Database size	Maximum	Build time

			distance	
Distance-to-conversion	3,438,146,250	232,848,000	26	30:32:34
Distance-to-mate	3,438,146,250	232,848,000	50	56:10:19

The program is currently running on a Pentium 133 PC with 32 MB memory under linux 1.2.13. Build time here is also includes a consistency check. Note that the choice of distance-to-conversion versus distance-to-mate makes little difference.

To solve any endgame, we have to solve all the subgames first. Both KAGP:KAAEE and KEGP:KAAEE have 71 subgames. Our program can generate all databases without human intervention. More detail on the program, databases, and the results about those endgame, are given in the first author's thesis. Here are the results with all subgames.

The Gödel coding scheme in our Chinese Chess program is considerably more complicated than the one used by most (western) chess programs. Our program has more than 13000 line of C code, compared to maybe 100 lines of C code in chess program.

Results for KAGP:KAAEE and all sub games.

Database	all	real	database	dis.-to-conv.		dis.-to-mate	
aehrgp-aehrgp				maxdis	time	maxdis	time
000000-000000	81	54	54	0	00:00:00	0	00:00:00
000000-010000	567	378	288	0	00:00:00	0	00:00:00
000000-020000	3969	2268	864	0	00:00:00	0	00:00:00
000000-100000	405	255	189	0	00:00:00	0	00:00:01
000000-110000	2835	1779	1242	0	00:00:00	0	00:00:00
000000-120000	19845	10638	3969	0	00:00:01	0	00:00:00
000000-200000	2025	948	342	0	00:00:00	0	00:00:00
000000-210000	14175	6588	2394	0	00:00:00	0	00:00:00
000000-220000	99225	39240	7182	0	00:00:00	0	00:00:00
000001-000000	4455	3015	2511	10	00:00:01	10	00:00:01
000001-010000	31185	20682	15840	2	00:00:01	6	00:00:02
000001-020000	218295	121554	47520	0	00:00:03	0	00:00:02
000001-100000	22275	13899	10395	6	00:00:01	7	00:00:01
000001-110000	155925	95004	68310	2	00:00:04	4	00:00:04
000001-120000	1091475	556362	218295	0	00:00:12	0	00:00:12
000001-200000	111375	50436	18810	2	00:00:01	5	00:00:01
000001-210000	779625	343332	131670	1	00:00:08	1	00:00:08
000001-220000	5457375	2002248	395010	1	00:00:24	1	00:00:24
000010-000000	7290	4914	4050	0	00:00:00	0	00:00:01
000010-010000	51030	33966	25920	0	00:00:03	0	00:00:03
000010-020000	357210	201204	77760	0	00:00:04	0	00:00:04
000010-100000	36450	22806	17010	0	00:00:01	0	00:00:01
000010-110000	255150	157110	111780	0	00:00:07	0	00:00:06
000010-120000	1786050	927540	357210	0	00:00:23	0	00:00:22
000010-200000	182250	83340	30780	1	00:00:02	1	00:00:02
000010-210000	1275750	571896	215460	1	00:00:17	1	00:00:15
000010-220000	8930250	3363048	646380	1	00:00:48	1	00:00:44
000011-000000	400950	270441	222750	7	00:00:57	10	00:01:04
000011-010000	2806650	1831884	1425600	20	00:08:49	21	00:07:30
000011-020000	19646550	10629810	4276800	12	00:10:15	22	00:12:11
000011-100000	2004750	1225887	935550	12	00:03:55	15	00:04:23
000011-110000	14033250	8274156	6147900	35	00:31:43	49	00:40:44
000011-120000	98232750	47838906	19646550	9	00:46:25	49	02:25:26
000011-200000	10023750	4374468	1692900	18	00:08:01	23	00:09:14
000011-210000	70166250	29403684	11850300	39	01:09:45	67	01:51:49
000011-220000	491163750	169290216	35550900	12	01:49:12	66	06:52:43
100000-000000	405	255	189	0	00:00:00	0	00:00:00
100000-010000	2835	1784	1280	0	00:00:00	0	00:00:00
100000-020000	19845	10698	3840	0	00:00:00	0	00:00:00
100000-100000	2025	1189	840	0	00:00:00	0	00:00:00
100000-110000	14175	8292	5520	0	00:00:00	0	00:00:00
100000-120000	99225	49566	17640	0	00:00:01	0	00:00:01
100000-200000	10125	4368	1470	0	00:00:01	0	00:00:00
100000-210000	70875	30348	10290	0	00:00:00	0	00:00:01
100000-220000	496125	180720	30870	0	00:00:02	0	00:00:02
100001-000000	22275	14163	10395	11	00:00:02	11	00:00:02
100001-010000	155925	97108	70400	2	00:00:08	6	00:00:08
100001-020000	1091475	570462	211200	0	00:00:11	0	00:00:11

100001-100000	111375	64553	46200	12	00:00:05	12	00:00:05
100001-110000	779625	441104	303600	2	00:00:19	4	00:00:19
100001-120000	5457375	2582394	970200	0	00:00:56	0	00:00:57
100001-200000	556875	231740	80850	2	00:00:05	8	00:00:05
100001-210000	3898125	1577220	565950	1	00:00:35	1	00:00:36
100001-220000	27286875	9196296	1697850	1	00:01:45	1	00:01:48
100010-000000	36450	22806	17010	9	00:00:06	9	00:00:05
100010-010000	255150	157540	115200	9	00:00:36	15	00:00:41
100010-020000	1786050	932640	345600	6	00:00:30	5	00:00:29
100010-100000	182250	104650	75600	11	00:00:21	14	00:00:23
100010-110000	1275750	720604	496800	9	00:00:55	3	00:00:36
100010-120000	8930250	4252320	1587600	2	00:01:52	2	00:01:57
100010-200000	911250	378332	132300	13	00:00:41	24	00:00:51
100010-210000	6378750	2595312	926100	4	00:01:23	8	00:01:31
100010-220000	44651250	15256536	2778300	2	00:03:29	2	00:03:36
100011-000000	2004750	1249157	935550	9	00:04:52	11	00:05:19
100011-010000	14033250	8456744	6336000	12	00:36:05	18	00:41:36
100011-020000	98232750	49044510	19008000	17	02:01:41	30	02:47:07
100011-100000	10023750	5604765	4158000	11	00:20:21	15	00:23:10
100011-110000	70166250	37813888	27324000	27	03:42:41	31	04:21:58
100011-120000	491163750	218538462	87318000	34	13:40:38	45	17:32:09
100011-200000	50118750	19805724	7276500	13	00:42:44	24	00:52:56
100011-210000	350831250	133085580	50935500	22	06:24:58	38	09:16:57
100011-220000	2455818750	765989832	152806500	32	19:30:42	47	28:13:30

Results for KEGP:KAAEE and all sub games.

Database	all	real	database	dis.-to-conv.		dis.-to-mate	
aehrgp-aehrgp				maxdis	time	maxdis	time
000000-000000	81	54	54	0	00:00:00	0	00:00:00
000000-010000	567	378	288	0	00:00:00	0	00:00:00
000000-020000	3969	2268	864	0	00:00:00	0	00:00:00
000000-100000	405	255	189	0	00:00:00	0	00:00:00
000000-110000	2835	1779	1242	0	00:00:00	0	00:00:00
000000-120000	19845	10638	3969	0	00:00:00	0	00:00:01
000000-200000	2025	948	342	0	00:00:00	0	00:00:00
000000-210000	14175	6588	2394	0	00:00:01	0	00:00:00
000000-220000	99225	39240	7182	0	00:00:00	0	00:00:00
000001-000000	4455	3015	2511	10	00:00:01	10	00:00:01
000001-010000	31185	20682	15840	2	00:00:01	6	00:00:01
000001-020000	218295	121554	47520	0	00:00:03	0	00:00:03
000001-100000	22275	13899	10395	6	00:00:01	7	00:00:01
000001-110000	155925	95004	68310	2	00:00:04	4	00:00:04
000001-120000	1091475	556362	218295	0	00:00:12	0	00:00:12
000001-200000	111375	50436	18810	2	00:00:01	5	00:00:01
000001-210000	779625	343332	131670	1	00:00:08	1	00:00:08
000001-220000	5457375	2002248	395010	1	00:00:24	1	00:00:24
000010-000000	7290	4914	4050	0	00:00:00	0	00:00:01
000010-010000	51030	33966	25920	0	00:00:03	0	00:00:02
000010-020000	357210	201204	77760	0	00:00:05	0	00:00:05
000010-100000	36450	22806	17010	0	00:00:01	0	00:00:01
000010-110000	255150	157110	111780	0	00:00:06	0	00:00:06
000010-120000	1786050	927540	357210	0	00:00:22	0	00:00:22
000010-200000	182250	83340	30780	1	00:00:02	1	00:00:02
000010-210000	1275750	571896	215460	1	00:00:14	1	00:00:14
000010-220000	8930250	3363048	646380	1	00:00:44	1	00:00:45
000011-000000	400950	270441	222750	7	00:00:58	10	00:01:04
000011-010000	2806650	1831884	1425600	20	00:07:00	21	00:07:30
000011-020000	19646550	10629810	4276800	12	00:08:23	22	00:12:09
000011-100000	2004750	1225887	935550	12	00:03:53	15	00:04:24
000011-110000	14033250	8274156	6147900	35	00:31:22	49	00:40:44
000011-120000	98232750	47838906	19646550	9	00:46:24	49	02:24:35
000011-200000	10023750	4374468	1692900	18	00:08:03	23	00:09:15
000011-210000	70166250	29403684	11850300	39	01:07:42	67	01:45:54
000011-220000	491163750	169290216	35550900	12	01:49:31	66	06:51:10
010000-000000	567	378	288	0	00:00:00	0	00:00:00
010000-010000	3969	2638	1984	0	00:00:01	0	00:00:00
010000-020000	27783	15780	5856	0	00:00:00	0	00:00:01
010000-100000	2835	1784	1280	0	00:00:00	0	00:00:00
010000-110000	19845	12412	8556	0	00:00:00	0	00:00:00
010000-120000	138915	74016	26880	0	00:00:02	0	00:00:02
010000-200000	14175	6628	2240	0	00:00:00	0	00:00:00
010000-210000	99225	45948	15680	0	00:00:01	0	00:00:01
010000-220000	694575	273000	47040	0	00:00:03	0	00:00:02

010001-000000	31185	20952	15840	10	00:00:03	10	00:00:04
010001-010000	218295	143324	109120	2	00:00:12	6	00:00:12
010001-020000	1528065	840000	322080	0	00:00:17	0	00:00:17
010001-100000	155925	96538	70400	12	00:00:08	12	00:00:08
010001-110000	1091475	658186	470580	2	00:00:29	4	00:00:28
010001-120000	7640325	3844548	1478400	0	00:01:24	0	00:01:25
010001-200000	779625	350104	123200	2	00:00:08	8	00:00:08
010001-210000	5457375	2377764	862400	1	00:00:53	1	00:00:54
010001-220000	38201625	13834176	2587200	1	00:02:41	1	00:02:42
010010-000000	51030	33966	25920	0	00:00:03	0	00:00:03
010010-010000	357210	234104	178560	0	00:00:19	0	00:00:19
010010-020000	2500470	1382772	527040	0	00:00:31	0	00:00:31
010010-100000	255150	157540	115200	1	00:00:07	1	00:00:07
010010-110000	1786050	1082446	770040	1	00:00:49	1	00:00:49
010010-120000	12502350	6373584	2419200	1	00:02:35	1	00:02:36
010010-200000	1275750	575296	201600	2	00:00:14	2	00:00:14
010010-210000	8930250	3938436	1411200	2	00:01:45	2	00:01:47
010010-220000	62511750	23104176	4233600	2	00:05:15	2	00:05:19
010011-000000	2806650	1855860	1425600	8	00:07:16	10	00:07:50
010011-010000	19646550	12537488	9820800	19	01:01:34	19	01:04:20
010011-020000	137525850	72555768	28987200	22	03:06:13	28	03:46:57
010011-100000	14033250	8407420	6336000	13	00:30:55	14	00:33:29
010011-110000	98232750	56606016	42352200	20	04:57:34	27	06:10:50
010011-120000	687629250	326463840	133056000	25	16:20:20	36	21:31:01
010011-200000	70166250	29980824	11088000	21	01:24:51	26	01:40:01
010011-210000	491163750	201064896	77616000	33	12:01:59	45	15:49:29
010011-220000	3438146250	1154955840	232848000	26	30:32:34	50	56:10:09

In the table, the database name here is represented by the number of each kind of pieces, for example, 100011-220000 means one Assistant, one Gunner and one Pawn against two Assistant and two Elephant. All is the number of all possible configurations of pieces in this endgame, even the illegal positions. Real is the number of positions which are legal (i.e. no two or more pieces sit in the same square). This is more human-useful when we talk about a particular endgame. Database is the size of the database for this endgame. We use several methods, including symmetry reduction and grouping etc. to reduce the size. This is described in detail in Wu's forthcoming thesis.

5. Conclusion

In this report, we present an improved retrograde algorithm which can be used to generate endgame databases. The algorithm requires less memory than earlier published algorithms, is faster, and also keeps full information about the endgame. It also offer the choose of using either distance-to-conversion or distance-to-mate without performance penalty.

The algorithm requires memory for only about 1 bit per position or 1/8 database size. This ratio just about matches a typical current computer's memory/disk. Considering that many consumer PCs now have 32MB memory and 1.6 GB disk space, most 5 men (western) chess endgame databases can be constructed in less than a day on such computers. We anticipate that it will not be long before the construction of 6-men chess endgame databases will be feasible on consumer hardware.

6. References

Clarke, M.R.B. (1977). A Quantitative Study of King and Pawn against King. *Advances in Computer Chess 1*, 108-118. Edinburgh University Press, Edinburgh.

Edwards, S. J. (1996). An Examination of the endgame KBBKN. *ICCA Journal*, Vol. 19, No. 1. 24-32.

Herik, H.J. and Herschberg, I.S. (1985). The Construction of an Omniscient endgame data base. *ICCA Journal*, Vol. 8, No.2, 66-87.

Lake, R. , Schaeffer J., and Lu, P. Solving Large Retrograde Analysis Problems Using a Network of Workstations. *Advances in Computer Chess 7,* Maastricht, Netherlands, 1994, 135-162.

Stiller, L.B. (1995). Exploiting Symmetry On Parallel Architectures. Ph.D. thesis. The John Hopkins University, Baltimore, Maryland.

Ströhlein, T. (1970). Untersuchungen über kombinatorische Spiele. Dissertation, Fakultät für Allgemeine Wissenschaften der Technischen Hochschule München.

Thompson, K. (1986). Retrograde Analysis of Certain Endgames. *ICCA Journal*, Vol. 9, No. 3. 131-139.

Wu, R. Some Research on Computer Chinese Chess. Ongoing Ph.D. thesis.